

Sistemas Operativos

Departamento de Computación - FCEyN - UBA

Primer cuatrimestre de 2024

Nombre y apellido: Lucas Di Salvo
Nº orden: 77 L.U.: 44418 Cant. hojas: 4

Primer parcial - 2do. cuatrimestre de 2024

1	2	3	Nota
25	35	25	35

25

A

- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido, LU y número de orden. Hojas sin identificar no serán corregidas.
- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma no se incluye en la cantidad total de hojas entregadas.
- Cada código o pseudocódigo debe estar bien explicado y justificado. ¡Obligatorio!
- Toda suposición o decisión que tome deberá justificarla adecuadamente. Si la misma no es correcta o no se condice con el enunciado no será tomada como válida y será corregida acorde.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen dos notas: I (Insuficiente): 0 a 64 pts y A (Aprobado): 65 a 100 pts.
- No difunda este enunciado.

Ejercicio 1.

La empresa not-AWS está probando un nuevo sistema para ejecutar programas en servidores en la nube. El sistema debe cumplir con las siguientes condiciones:

- Por defecto todos los programas tienen la misma prioridad. Pero cada tanto, por un breve período de tiempo, un programa puede utilizar el CPU más que otros con el fin de terminar lo más rápido posible su tarea.
- Los programas que realizan mucho cómputo por un tiempo prolongado deben reducir su prioridad para permitir que se ejecuten otros programas.
- Un usuario tiene la posibilidad de pagar extra para ejecutar sus programas de manera urgente, dentro de una cota de tiempo baja.
- El SLA (acuerdo de nivel de servicio) establece que no puede haber programas que queden sin ejecutarse.
- Mantener programas en el sistema durante mucho tiempo genera problemas. Por eso, los programas que realizan poco cómputo durante mucho tiempo deben aumentar su prioridad.
- Si un programa varía su consumo de poder de cómputo, las reglas antes descritas deben adaptarse dinámicamente para cada momento del programa.

Dado que el scheduler por defecto no cumple con estas reglas, se necesita un nuevo algoritmo. Diseñar una solución de scheduling que garantice este comportamiento. Justificar las decisiones tomadas y explicar si el sistema puede o no tener starvation.

Ejercicio 2.

Modelar un sistema de trenes concurrentes, sin condiciones de carrera o deadlocks que cumpla con:

- Hay N trenes numerados del 0 al N-1 que recorren el trayecto de 10 estaciones.
- Cada tren tiene una capacidad máxima de M pasajeros. Los pasajeros esperan en cada estación de partida hasta que llegue un tren con asientos disponibles.
- Cuando un tren llega a una estación, los pasajeros se suben aplicando la función `abordarTren()` hasta llenar el tren o hasta que no haya más pasajeros en la estación.
- Una vez que el tren esté lleno o no queden más pasajeros esperando, el tren parte hacia la siguiente estación. Se tiene que llamar a la función `viajar()`.
- En la última estación todos los pasajeros bajan del tren con la función `descenderTren()` y, cuando todos han descendido, el tren inicia el recorrido en el sentido contrario para repetir el proceso.

Adicionalmente, considerar las restricciones:

- No puede haber dos trenes en la misma estación en el mismo sentido al mismo tiempo. Si un tren está cargando o descargando pasajeros, cualquier otro tren en el mismo sentido debe esperar hasta que la estación esté disponible.
- Los trenes deben salir en orden secuencial: primero el tren 0, luego el tren 1, y así sucesivamente hasta el tren N-1, y no pueden adelantarse.
- Puede haber múltiples trenes viajando al mismo tiempo.
- Por simplicidad en esta versión inicial del sistema se puede modelar que los pasajeros no podrán entrar a la estación mientras se está abordando un tren.
- Utilizar las funciones *descenderTren*, *abordarTren*, y *viajar*.

Solo se permiten usar las primitivas de sincronización vistas en la teórica.

Ejercicio 3.

Dado el siguiente código:

- Siga los cambios de variables e indique correctamente en cada comentario qué proceso ejecuta ese bloque de código (padre/hijo/etc) y el valor de la variable `globalCounter`.
- Enumere los problemas en este código. Justifique cada caso.
- Corrija y extienda el código para solucionar los problemas encontrados. Además, utilizando señales, y después de que todos los hijos hayan terminado, imprimir en pantalla desde el padre el siguiente mensaje: "Se fueron los chicos, hora de un buen mate."

```
int globalCounter = 5;
```

```
int main() {
    pid_t pid1, pid2;
    globalCounter += 5;
    pid1 = fork();

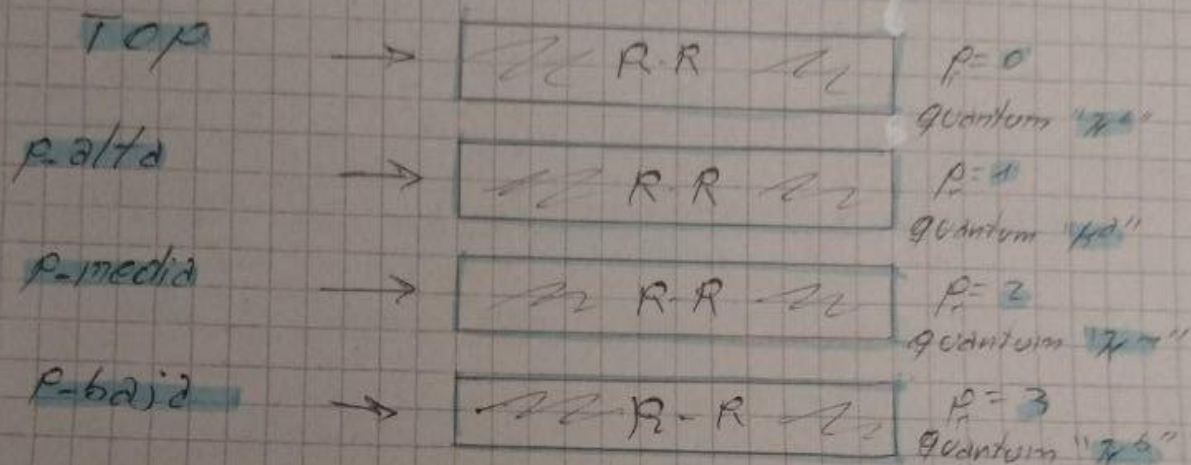
    if (pid1 == 0) {
        globalCounter += 10;
        pid2 = fork();

        if (pid2 == 0) {
            globalCounter += 20;
            // Seleccione Soy el padre / Soy el 1er hijo / Soy el 2do hijo / Soy el 3er ①
            hijo
            printf("globalCounter = %d", globalCounter); // globalCounter = [ ] ?
            exit(0);
        } else if (pid2 > 0) {
            globalCounter += 5;
            // Seleccione Soy el padre / Soy el 1er hijo / Soy el 2do hijo / Soy el 3ero ②
            hijo
            printf("globalCounter = %d", globalCounter); // globalCounter = [ ] ?
            exit(0);
        }
    } else if (pid1 > 0) {
        wait(NULL);
        globalCounter += 2;
        // Seleccione Soy el padre / Soy el 1er hijo / Soy el 2do hijo / Soy el 3ero ③
        hijo
        printf("globalCounter = %d", globalCounter); // globalCounter = [ ] ?
        sleep(5);
    }

    return 0;
}
```


Ejercicio 1:

El algoritmo de scheduling que propongo consiste de 4 colas de prioridad, donde cada una de ellas utiliza una política Round-Robin.



- Todos los procesos comienzan ingresando a la cola P-media con quantum π^2 . Al consumir su quantum, un proceso de esta lista, se evalúa si su ráfaga remanente (r'), resulta menor a π^2 , en cuyo caso, se mantiene en P-media, caso contrario, baja a P-baja. Adicionalmente, si un proceso se halla en P-alta, se realiza la misma comparación, y si $r' > \pi^2$, pasa a P-media. En cualquiera de estos casos, pasar al fondo de la cola correspondiente.

- Adicionalmente, un cliente puede solicitar un "upgrade" en la prioridad de los procesos a su nombre (dichos procesos presentes en P-alta, P-media o P-baja), en cuyo caso, los procesos

distinguidos pasan a la cola Top. /

- Por último, es relevante mencionar que si un proceso en p -baja pasa mucho tiempo en la misma, puede subir de prioridad a p -media (idem entre p -media y p -alta), y que a menor número de prioridad más "importante" es la cola. Este modelo se lo conoce como multilevel feedback queue. → aguing.

- Tomo que $\tau^L < \tau^e < \tau^m < \tau^h$. /

- Dado que los procesos pueden cambiar su prioridad en caso de ser necesario, y junto con que las colas usen política R.R., este modelo carece de inanición, como estipula el SLA del enunciado.

- ¿ Los procesos de TOP pueden bajar de prioridad? /
- Si no → ¿ hay starvation de los no-top? /
- Si sí → ¿ garantizás el privilegio de pago? /
-

Ejercicio 2:

Voy a contar con las siguientes variables globales:

- estaciones = [..., [sem(1), sem(2)], ...]

la cual es un array bidimensional de semáforos, que distingue a cada estación (10) en un primer nivel y su sentido en el segundo.

- trains = [sem(1), sem(0), ..., sem(0)], que permite que los trenes funcionen en orden.

- estacion_mutex = [..., [sem(1), sem(1)], ...] el cual es un mutex por estación (y sentido) para sincronizar arribos de pasajeros y trenes.

- sem_abordar = [..., [sem(1), sem(1)], ...] que permite que pasajeros que ya llegaron a la estación previo a un tren, suban al tren que acaba de llegar (10 estaciones con sus 2 sentidos).

- pasajeros = [..., [0, 0], ...] un array de 10 estaciones y sus 2 sentidos que cuenta la cantidad de pasajeros por estación (estos contadores son protegidos por estacion_mutex).

- sem_ultima_estacion = sem(0), que denota si se llegó a la última estación. Contoso

Consideraciones:

- Supongo que cada tren conoce su número n (lo pasa por parámetro) y la capacidad M del enunciado. Supongo que cada pasajero conoce la estación y sentido a la cual arriba (los paso por parámetro).
- Por otro lado, frente a consultas realizadas durante el parcial, concluir que un pasajero no sabe a qué número de tren se subió. De forma que no me preocupo por el que quien descendiendo del tren frente a que un tren avisó haber arribado en la última estación según efectivamente sus pasajeros. La verdad esto se maneja con un array pero como no le afecta la nota

Código

// Pasajero

$P(i, s):$

// estación y sentido

estacion - mutex $[i][s]$. wait(); // molinete

pasajeros $[i][s]++$;

// nuevo pasajero en i, s

estacion - mutex $[i][s]$. signal();

// permite que pase otra persona o llegue el tren

Sem - abordar $[i][s]$. wait();

// espera para subir

aboardarTren();

// sube

Sem - ultima - estacion. wait();

// es para la última estación

descenderTren();

// baja

// tren

 $T(n)$

// número de tren

Sentido = 0; capacidad = M;

trenes $[n]$. wait();

// puedo arrancar

While (true)

// cuerpo a ejecutar

for (i = 0 to 8) do

// recorro las estaciones

estacion $[i]$ [sentido]. wait(); // espero para arribar

if (i = 1) then

// si es segunda estación

trenes $[(n+1) \% N]$. signal(); // le aviso al siguiente

// tren que arranque

estacion_mutex $[i]$ [sentido]. wait();While (pasajeros $[i]$ [sentido] > 0 &&
capacidad > 0)// hay pasajeros
// que pueden subirpasajeros $[i]$ [sentido] --;

capacidad --;

sem_abordar $[i]$ [sentido]. signal();

// suben pasajeros

estaciones $[i+1]$ [sentido]. wait();

viajar();

estaciones $[i]$ [sentido]. signal();

sem_ultima_estacion. signal (M - capacidad);

estaciones $[i+1]$ [sentido]. signal(); // libero última

Sentido = (sentido + 1) % 2; // cambio sentido

Buen, falta diferenciar pasajeros por tren,
pero entiendo que pudo haber controversia
durante el parcial así que si te computaba
no te afecta la nota para este ítem

Ejercicio 3:

- a) - En el comentario ①, el valor de `globalCounter` será 40, y dicho bloque de código lo ejecutará el hijo 2, pues en la guarda previa se evalúa si `pid2 == 0`, y dicho valor es cero dentro del contexto de ejecución del proceso creado por el segundo `Fork()`.
- En el comentario ②, el valor de `globalCounter` será 25, pues nos encontramos en el contexto de ejecución del primer hijo (pues en el padre, al crear un proceso hijo el `pid` obtenido será distinto de cero).
- En el comentario ③, el valor de `globalCounter` será 12, pues como en el comentario previo, no ingresamos a la guarda correspondiente al hijo (con `pid1`), sino que seguimos la rama del padre. B/
- b) - Si bien puede ser el comportamiento deseado, la variable `globalCounter` no se comparte, i.e., si bien la PBC se copia al clonar, no se actualiza dinámicamente.
- Por otro lado, y aún más relevante, no se garantiza la terminación en orden de los hijos (por ejemplo, el primer hijo puede terminar antes que el segundo). B/
- También se podrían utilizar otros mecanismos de sincronización (`kill` o `signal`) en vez de utilizar un `sleep`.

c) Por un lado la guarda del primer hijo
(i.e. $\text{else if } (\text{pid} > 0)$) podría incluir
★ ~~sleep(5)~~ para asegurar que efectivamente
finalice la ejecución del segundo hijo.

~~Por otro lado el padre antes de~~

★ : `signal (hijo1, SIGCHLD);`
`while (true);`

★ `Hijo1 (sig):`

`exit(EXIT_SUCCESS);` x no estamos aquí

Por otro lado, de manera similar,
el padre podría ~~eliminar~~ el `wait(NULL)`,
~~no tener~~ eliminar `sleep(5)`, ~~se~~ e incluir
antes del `return 0`

`signal (padre, SIGCHLD)` ~~on~~
~~return 0;~~

`while (true):`

`return 0;`

x FALTAN soluciones
que los hijos se "esperen"

donde:

`padre (sig):`

`printf ("se fueron los chicos,`
`hora de un buen mate.");` ~~on~~

`return 0;`